Foundations of Test on z/OS

Michael Gildein

1918

Agenda

Test Basics

- What is Test? and Why Test?
- IBM Z Stability Requirements
- Perfect Verification Paradox
- Principles of Testing

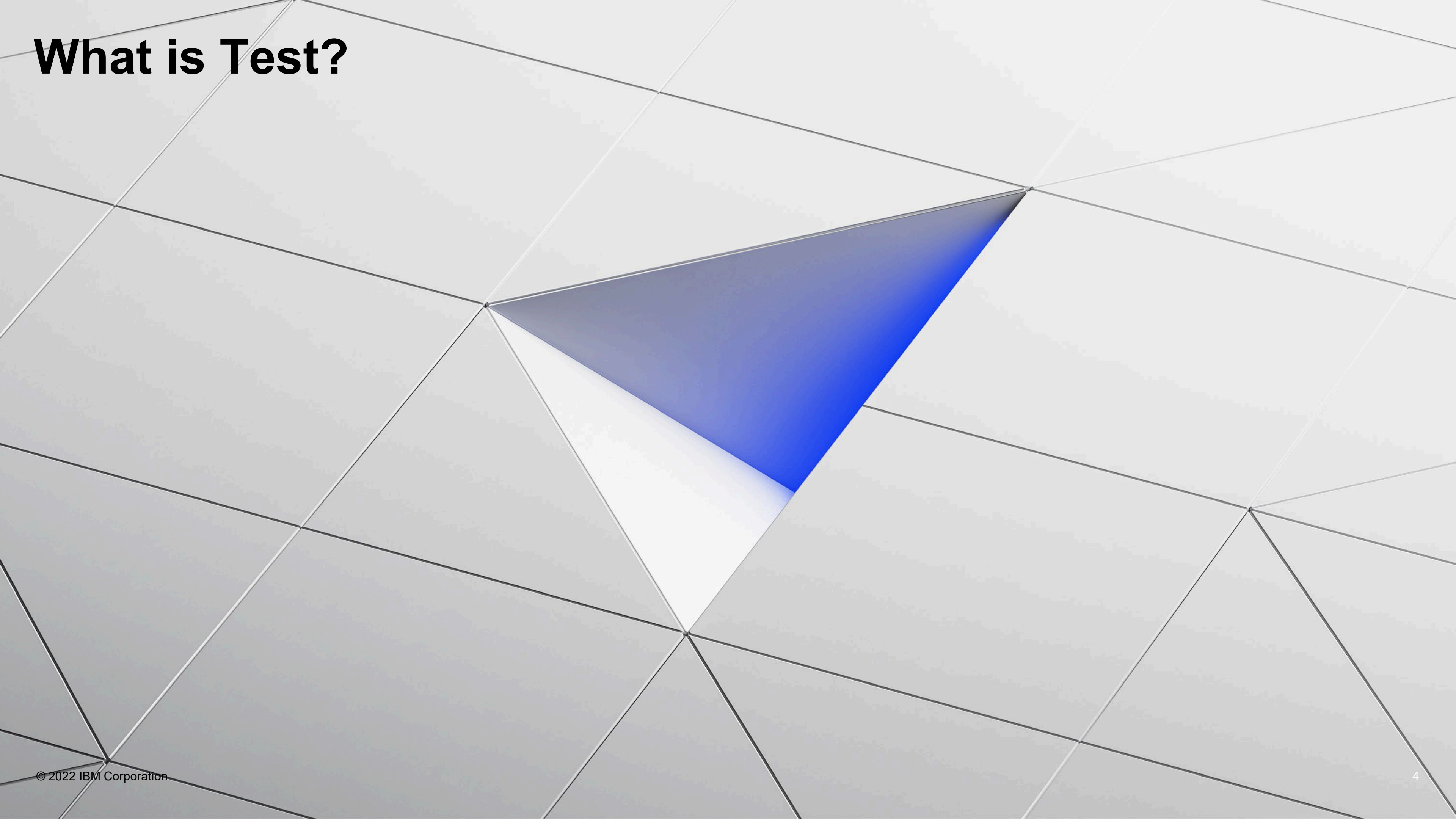
Test Classification

- Verification, validation, vs testing
- Test Types
- Static vs Dynamic

Test Phases

- Unit Test
- Function Test
- System Test
- Acceptance (Platform) Test

What is Test?



What is Test?

“A procedure intended to establish the quality, performance, or reliability of something, especially before it is taken into widespread use.”

Purpose

- Remove defects
- Validate conformity to requirements (not specification!)

What is a Defect?

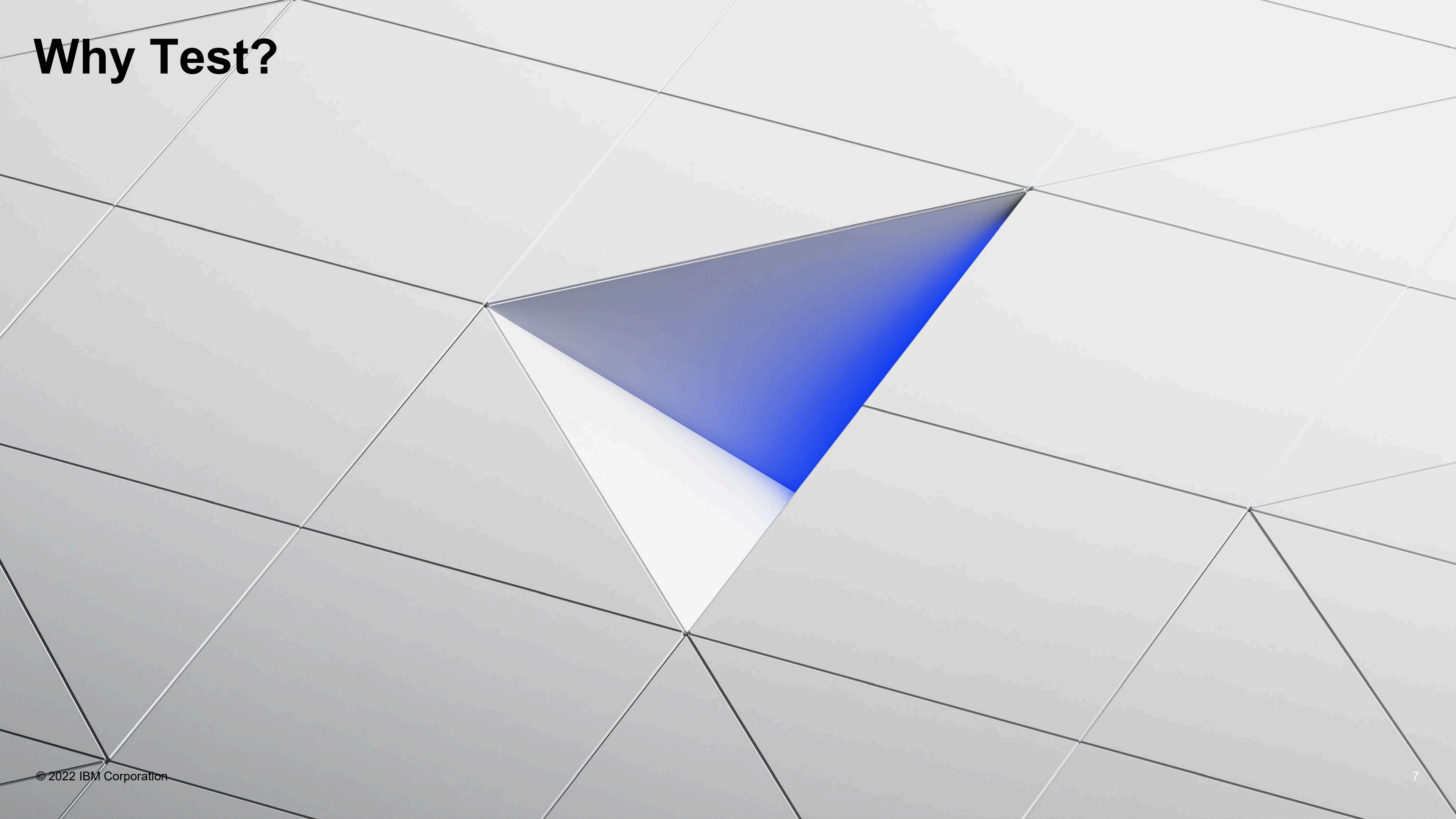
“Problem or flaw in a program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways.”

error = defect = bug

Fault is the manifestation or an instance of one or more defects during execution.

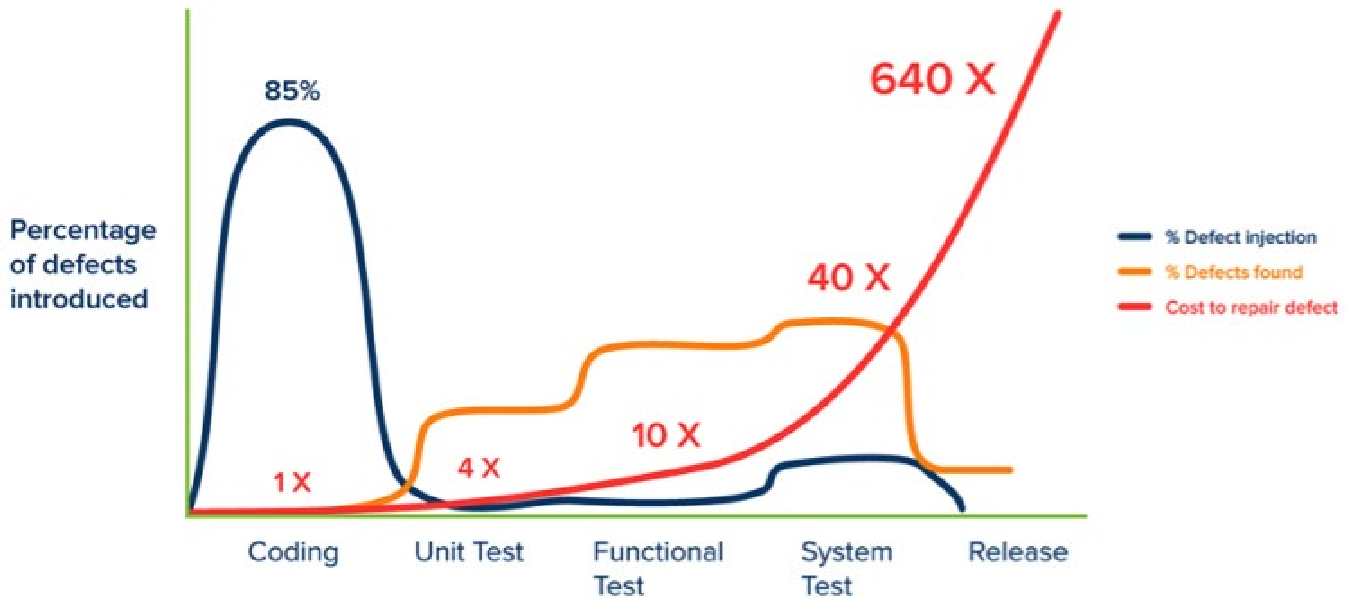
Failure occurs when a fault produces an undesired state that may propagate to the programs output or behavior.

Why Test?



Why Test?

Defects have impact in terms of cost, reputation/trust, and legal issues.



Jones, Capers. *Applied Software Measurement: Global Analysis of Productivity and Quality.*

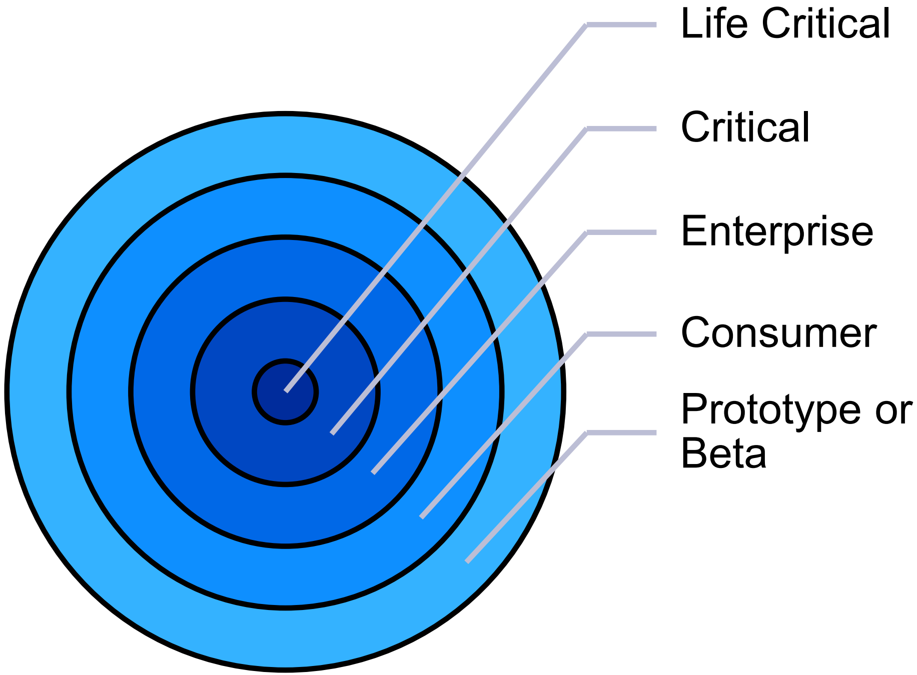
https://www.stickyminds.com/sites/default/files/shared/2018-12-10%20ArthurHicken%20The%20Shift-Left%20Approach%20to%20Software%20Testing%20image3.png

Stability Requirements

Stability Requirements

Classes of products

- Prototype or Beta
  - As-is; defects expected; no warranty/SLAs
- Consumer
  - Minimal cost; Fixes not guaranteed;
  - Infrequent defects expected; limited impact
- Enterprise/Industrial
  - 99.99%+ uptime; high costs
  - No defects expected; impact large userbase
- Mission | Business Critical
  - ~100% uptime; SLAs; DR w/ failover
  - Defects can introduce market shift & revenue loss
- Life Critical
  - 100%+ uptime/reliability; multiple levels of recovery
  - Failure is unacceptable; results in injury or loss of life



Service Level Agreement (SLA) - Commitment about aspects of quality, availability, responsibilities between a server provider and consumer.

Disaster Recovery (DR) - Set of policies, tools and procedures to enable the recovery or continuation of critical infrastructure/systems following a disaster.

Stability Requirements

99.99999% availability equates to < 3 seconds downtime per year!

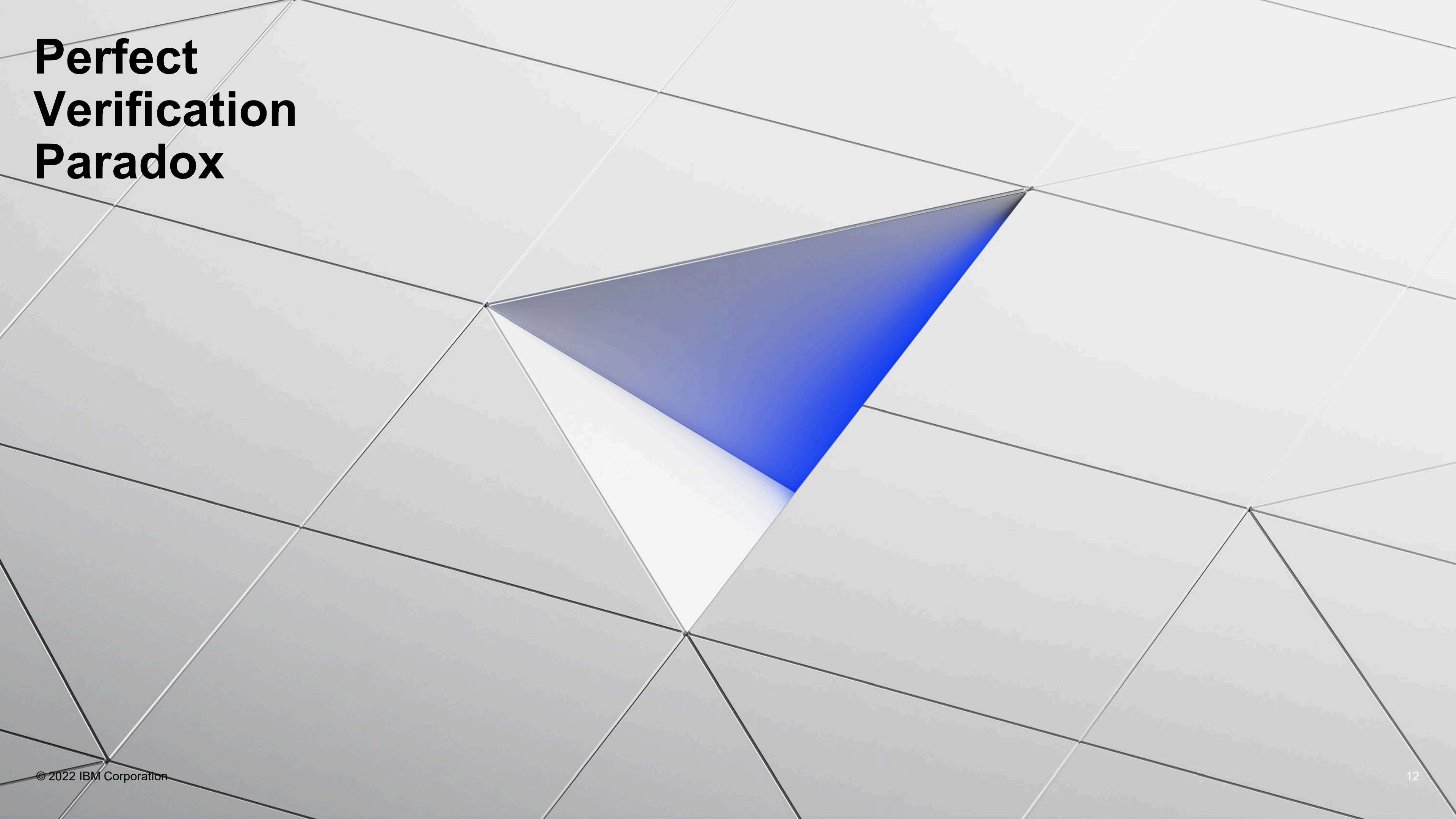
Reliability Availability Serviceability (RAS)
Quality Metrics

- Unplanned outages
- Client impacting events
- Repair actions

Clients expect flawless full stack operation

Solution Testing
Platform Evaluation Testing (zPET)
Running middleware and client like workloads

Perfect Verification Paradox



Perfect Verification Paradox

- Real world problems are complex
- Ever increasing development process complexity
- Solutions naturally become more complex as they evolve without overt action
- Client/user functional expectations continually increase
- Quality is perceived as declining unless rigorously maintained

Why not just test everything?

Perfect Verification Paradox

Exhaustive Testing

Test all possible combinations of configurations, parameters, and inputs.

Browsers

- Microsoft Edge
- Google Chrome
- Apple Safari
- Mozilla Firefox

Operating Systems

- Microsoft Windows
- Google Android
- Apple iOS

Accessibility

- Mouse
- Keyboard
- Touchscreen
- Screen readers
- Night mode

Perfect Verification Paradox

Exhaustive Testing

Test all possible combinations of configurations, parameters, and inputs.

Browsers

- Microsoft Edge
- Google Chrome
- Apple Safari
- Mozilla Firefox

Operating Systems

- Microsoft Windows
- Google Android
- Apple iOS

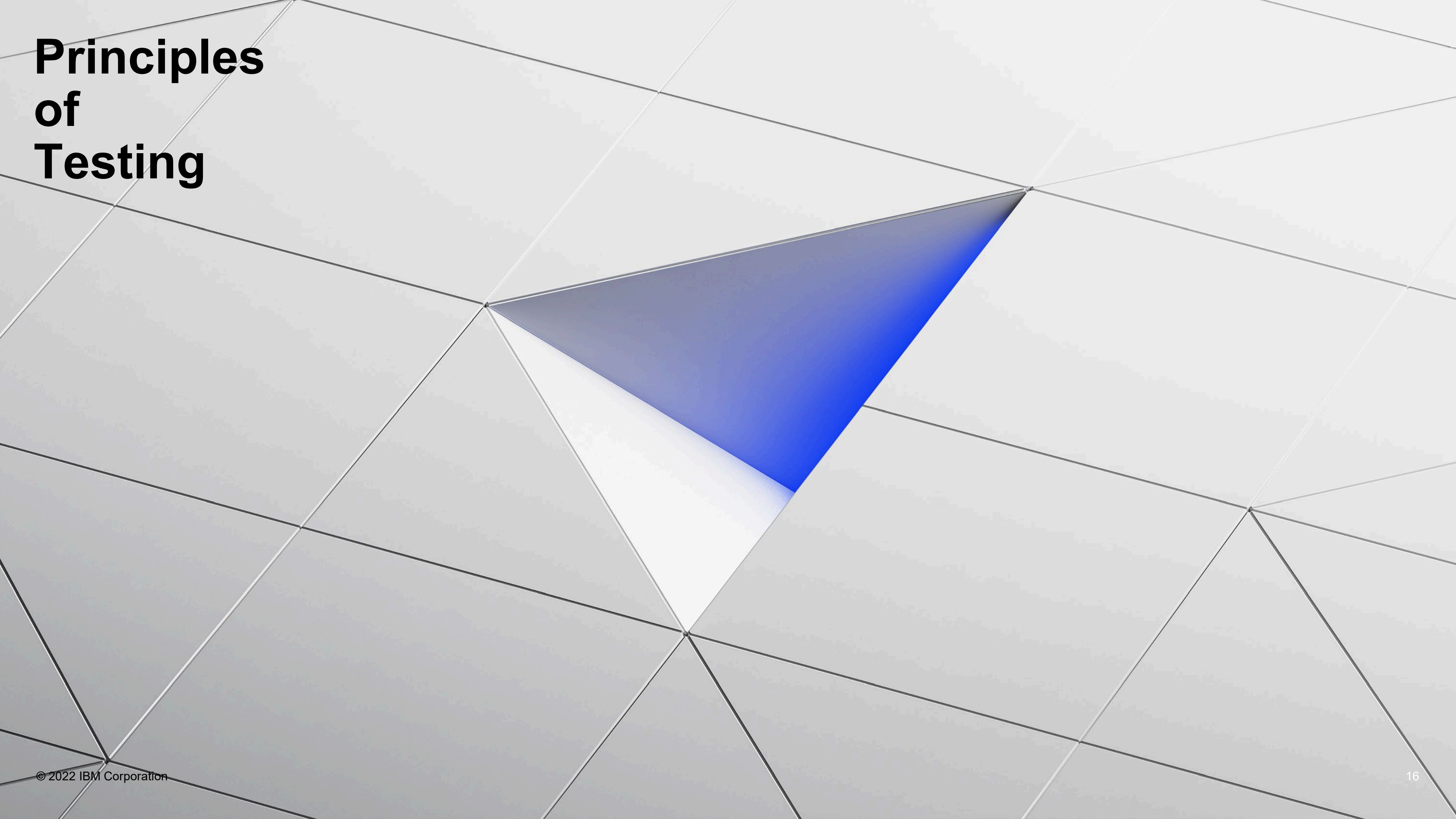
Accessibility

- Mouse
- Keyboard
- Touchscreen
- Screen readers
- Night mode

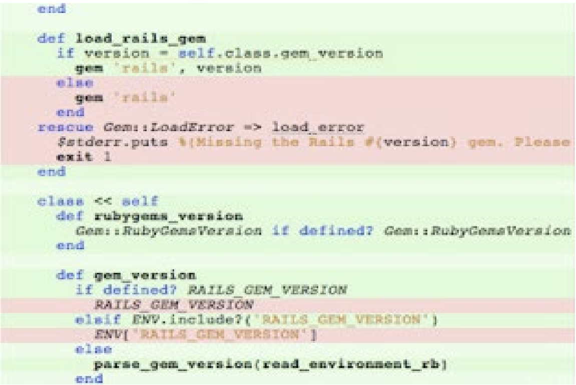
4 * 3 * 5 = 60

What about browser levels, screen resolutions,
… for each page and function?

Principles of Testing



Principles of Testing



```
end

def load_rails_gem
  if version = self.class.gem_version
    gem 'rails', version
  else
    gem 'rails'
  end
rescue Gem::LoadError => load_error
  $stderr.puts %(Missing the Rails #{version} gem. Please
  exit 1
end

class << self
  def rubygems_version
    Gem::RubyGemsVersion if defined? Gem::RubyGemsVersion
  end

  def gem_version
    if defined? RAILS_GEM_VERSION
      RAILS_GEM_VERSION
    elsif ENV.include?('RAILS_GEM_VERSION')
      ENV['RAILS_GEM_VERSION']
    else
      parse_gem_version(read_environment_rb)
    end
```

Testing shows presence of defects

Not the absence

Reliability and Confidence

Testing may increase one’s confidence in the correctness of a program though the confidence may not match with the program’s reliability

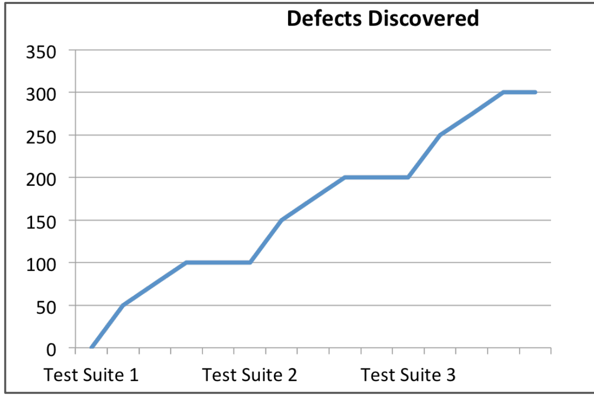
Coverage

- A test case that tests untested portions of a program enhances or diminishes one's confidence in the program’s correctness depending on whether the test passes or fails
- Code coverage is a reliable metric for the quality of a test suite

Requirements

Tests derived manually from requirements alone are rarely complete

Principles of Testing



Pesticide paradox

Process of repeating the same test cases again and again, eventually, the same test cases will no longer find new bugs

Randomness

Random testing may or may not outperform non-random testing

Saturation Effect

The saturation effect or defect discovery rate is real and can be used as an effective tool of test generation strategies

Roles

Every developer
…is also a tester

- Testability
- Static code analysis
- Defensive Programming
- Unit test

Roles

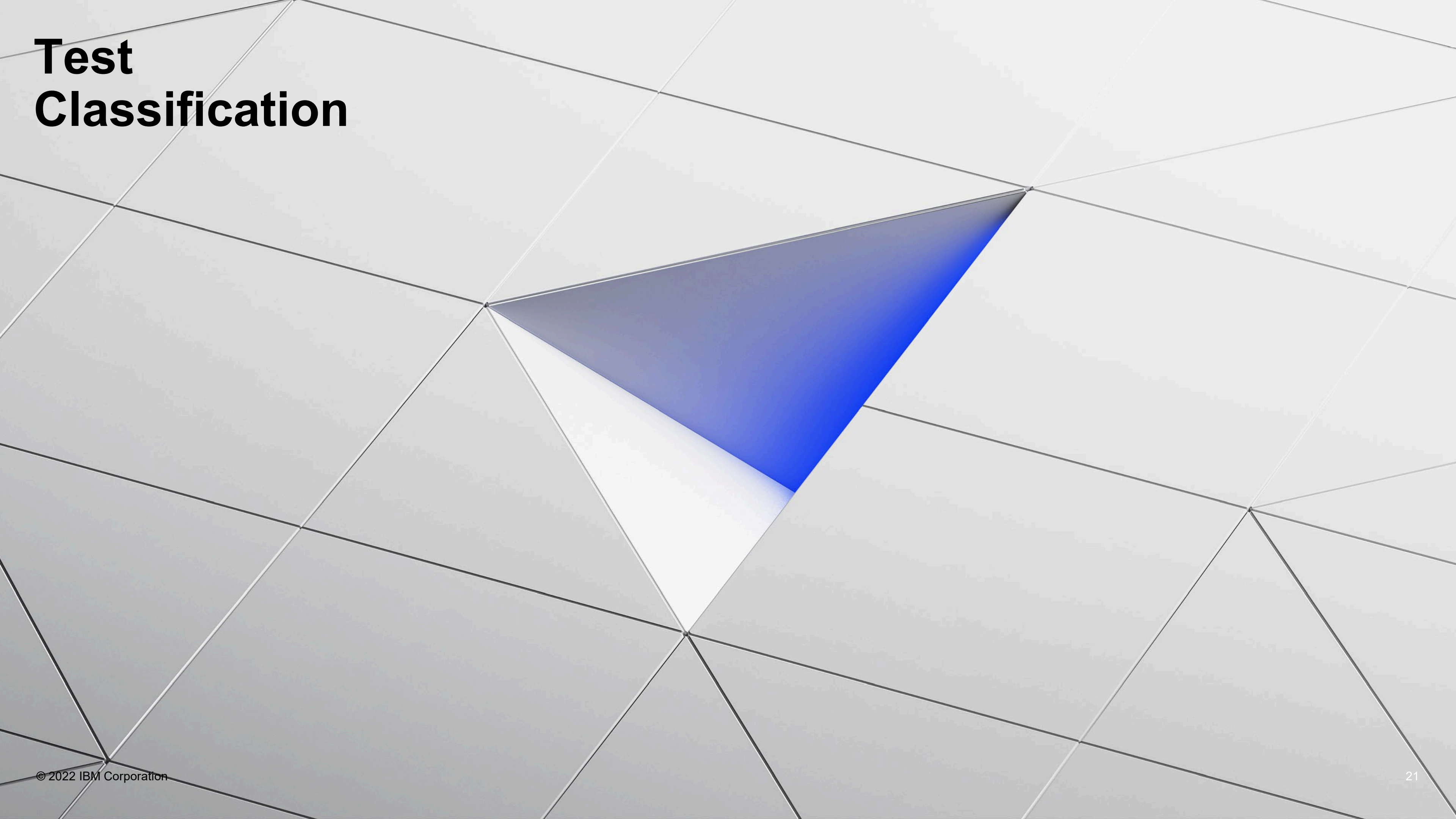
Every developer
…is also a tester

- Testability
- Static code analysis
- Defensive Programming
- Unit test

Tester Characteristics

- Curious/likes asking questions
- Likes problem solving/creating
- Detail oriented
- Outspoken/good communication skills
- Patient
- Likes scavenger hunts/debugging
- Thinks outside the box
- Dedicated
- Persevering
- Cares about the product
- Works well in a team
- Likes tinkering
- Likes coding
- Creative

Test Classification



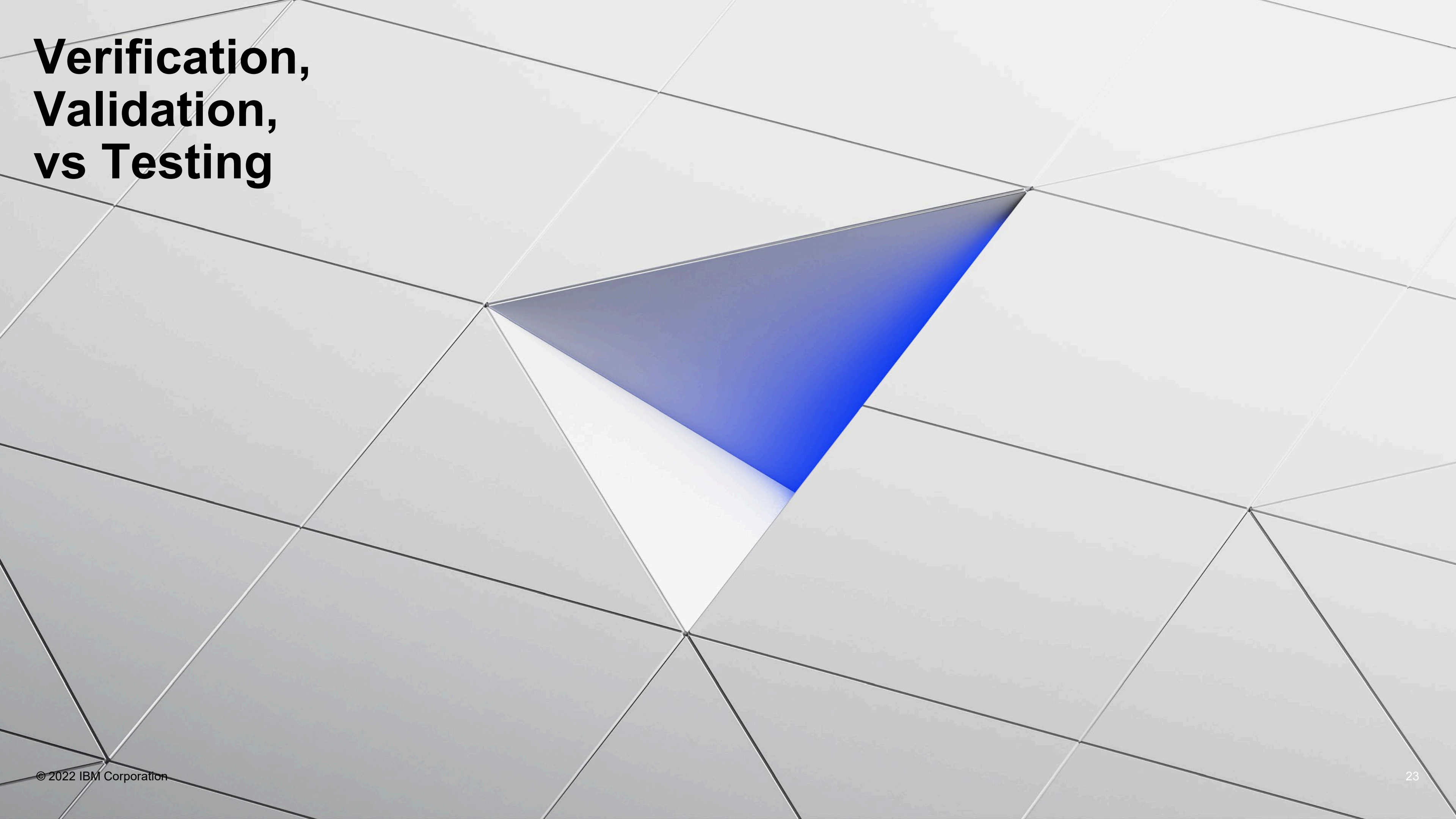
Test Classification

- Dividing the test domain space into logical units
- Helps segment the domain into schedulable, manageable, and containable units
- Develop experts in specific areas faster than all of testing

Classify by

- **Technique**
  Test execution or generation techniques
- **Phase**
  Successive compounding test stages
  w/ specific env and focus
- **Goal**
  Desired end result
  i.e., security

Verification, Validation, vs Testing



Verification, Validation, vs Testing

Verification

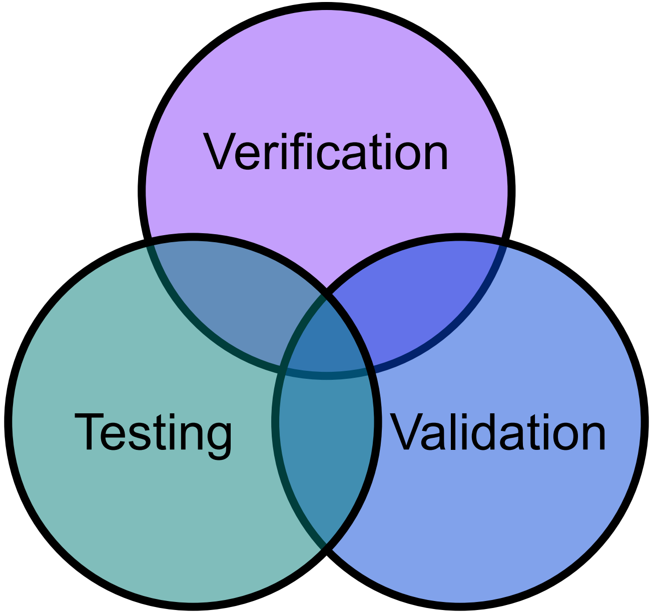
- "A test of a system to prove that it meets all its specified requirements at a particular stage of its development." – IEEE-STD-610
- Prove correctness of a program based on specification and requirements
- Test to pass || "Happy" or "Good" path testing

Validation

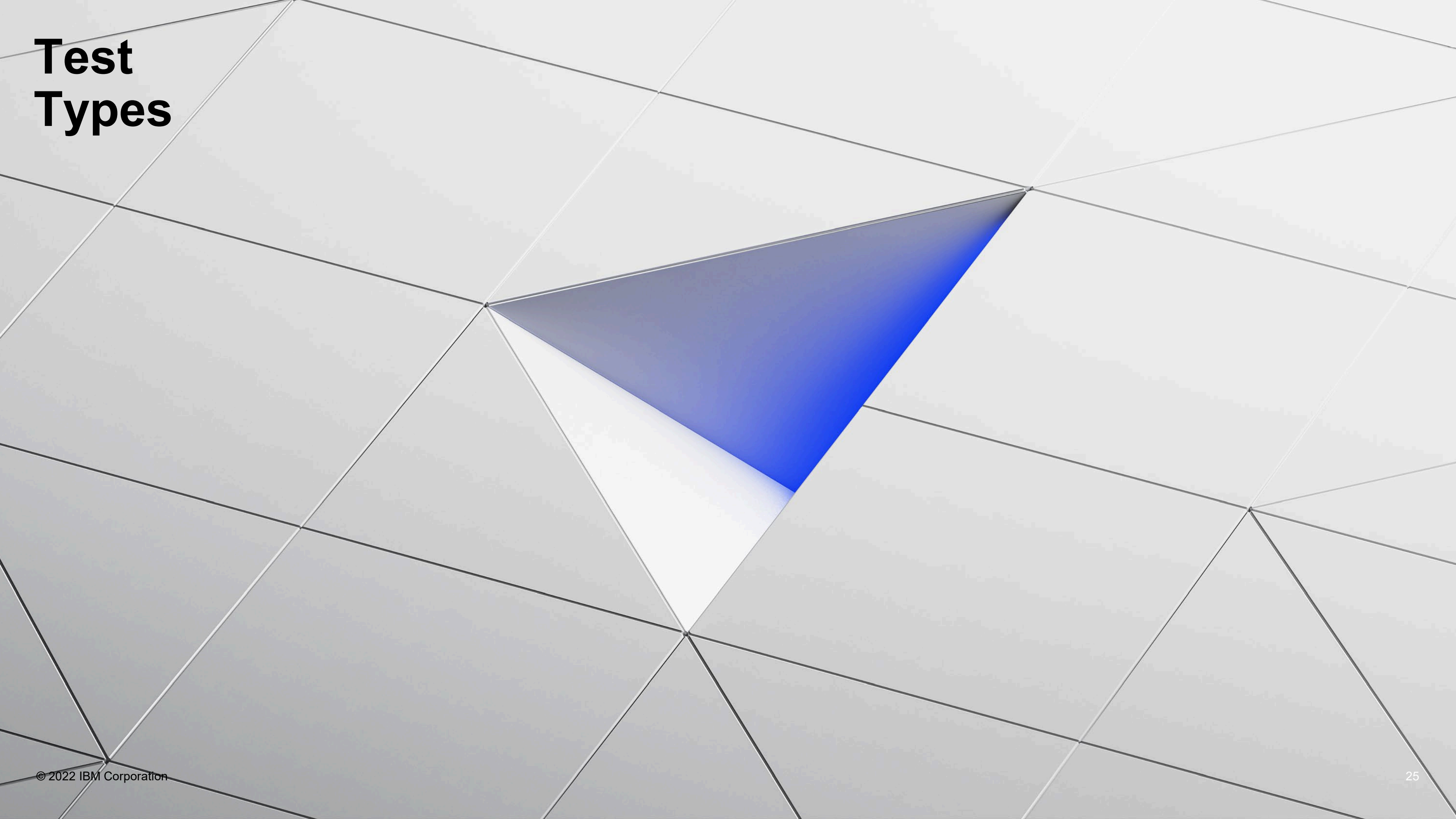
- "An activity that ensures that an end product stakeholder's true needs and expectations are met." – IEEE-STD-610
- Also known as acceptance or business testing

Testing (Destructive)

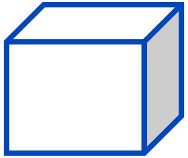
- Discover defects
- Test to fail
- Malicious intent
- "Bad" path testing



Test Types

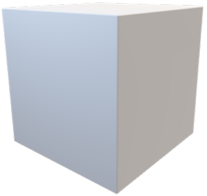


Box Type Classification



White/Clear Box

- Based on logic and internals such as code structure
- Developers' perspective
- Tester, can see code as its executing
- Easy debugging



Gray Box

- Based on requirements
- Enhanced with knowledge about code and implementation
- Very common

Black Box

- Behavioral testing
- Can only see inputs/outputs
- Based on requirements and documentation
- Customer perspective
- Code is executed but not used to create or enhance tests

Functional vs Structural

Functional

- Program behaves compared to requirement specifications
- What the program does
- Type of black box

Structural

- Program behaves compared to intention of programmer
- How the program does it
- Type of white box

Functional Examples

- Regression
- Usability
- Behavioral
- ...

Structural Examples

- Statement Coverage
- Branch Coverage
- Path Coverage
- ...

Non-functional Tests

- Performance
- Scalability
- Security
- Constraints
- Installation
- Migration
- Co-existance
- Compatibility
- …

*These may also be defined as requirements

Static vs Dynamic Technique

Static

- Program/System not executed
- Inexpensive

Examples

- Code inspection/reviews
- Document review
- Intellectual Property (IP) scans
- Complexity analysis
- Security scans
- Coding standards & patterns

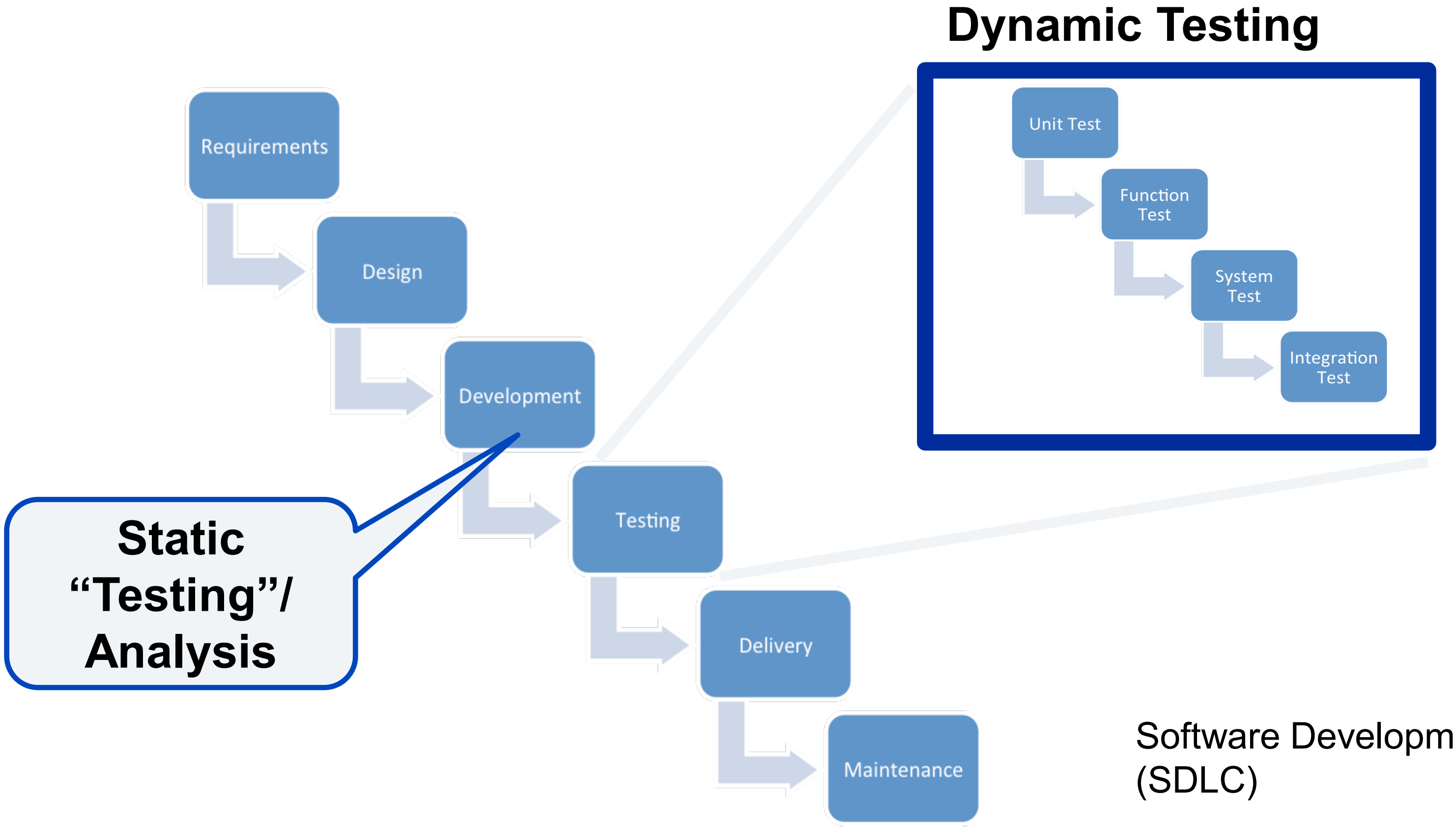
Dynamic

- Program/system is executed
- Expensive

Examples

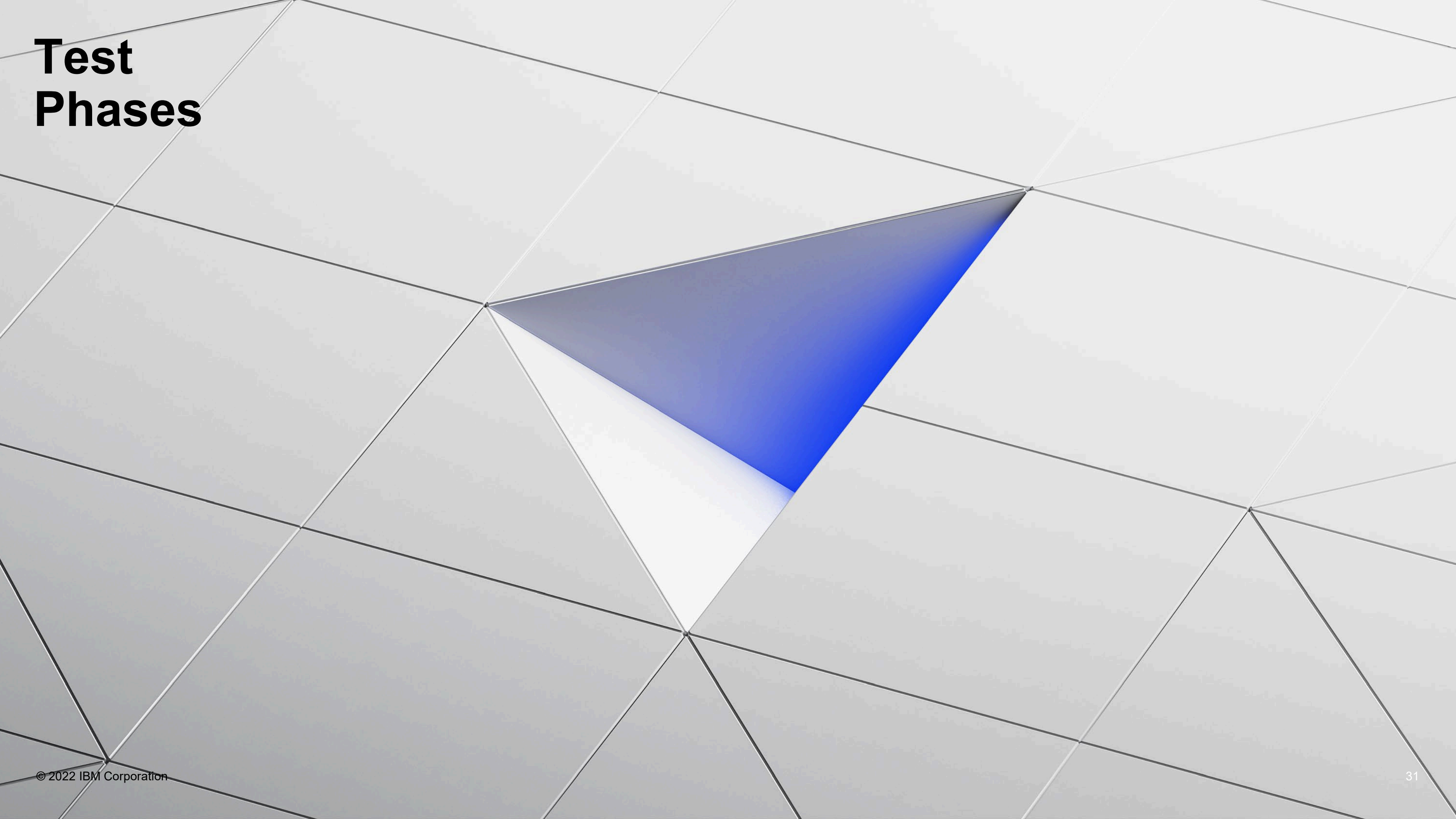
- Classic testing
  - Regression
  - Load/Stress
  - Phases

Waterfall SDLC Model

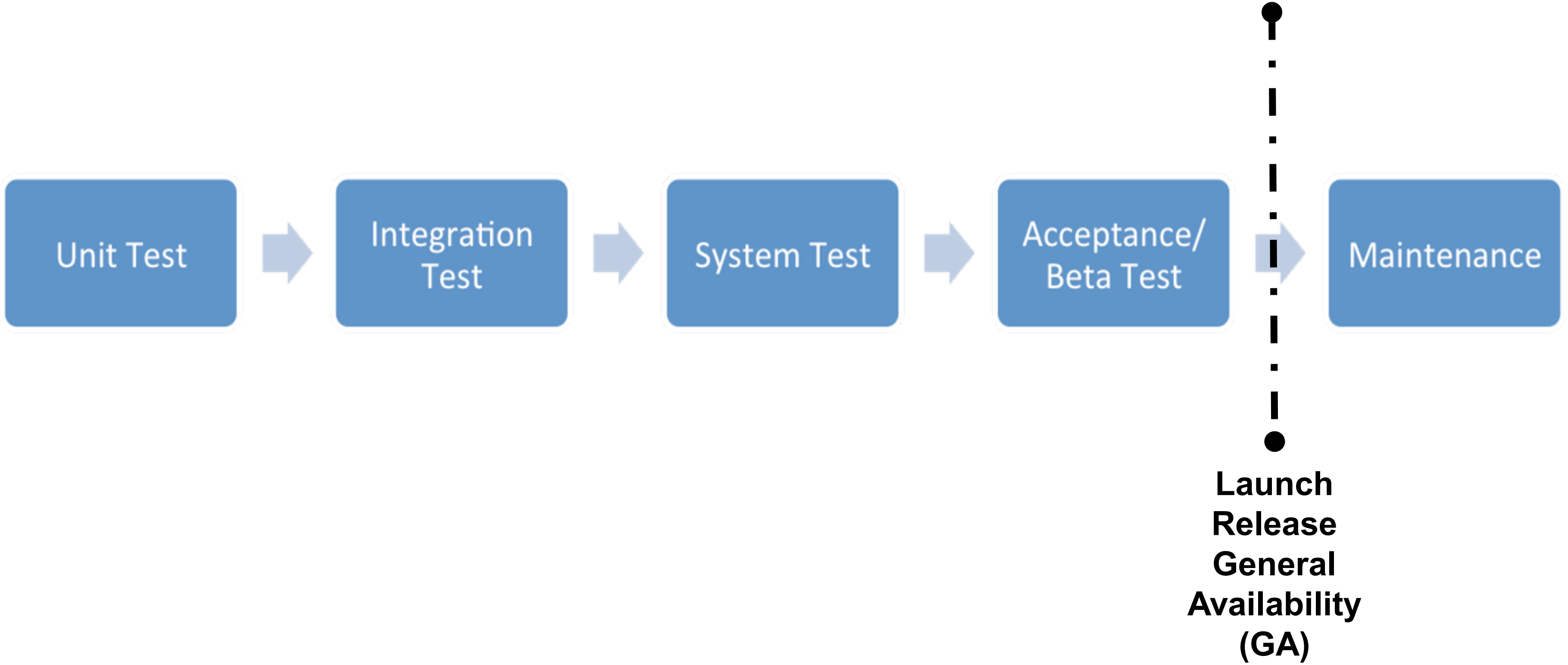


Software Development Life Cycle (SDLC)

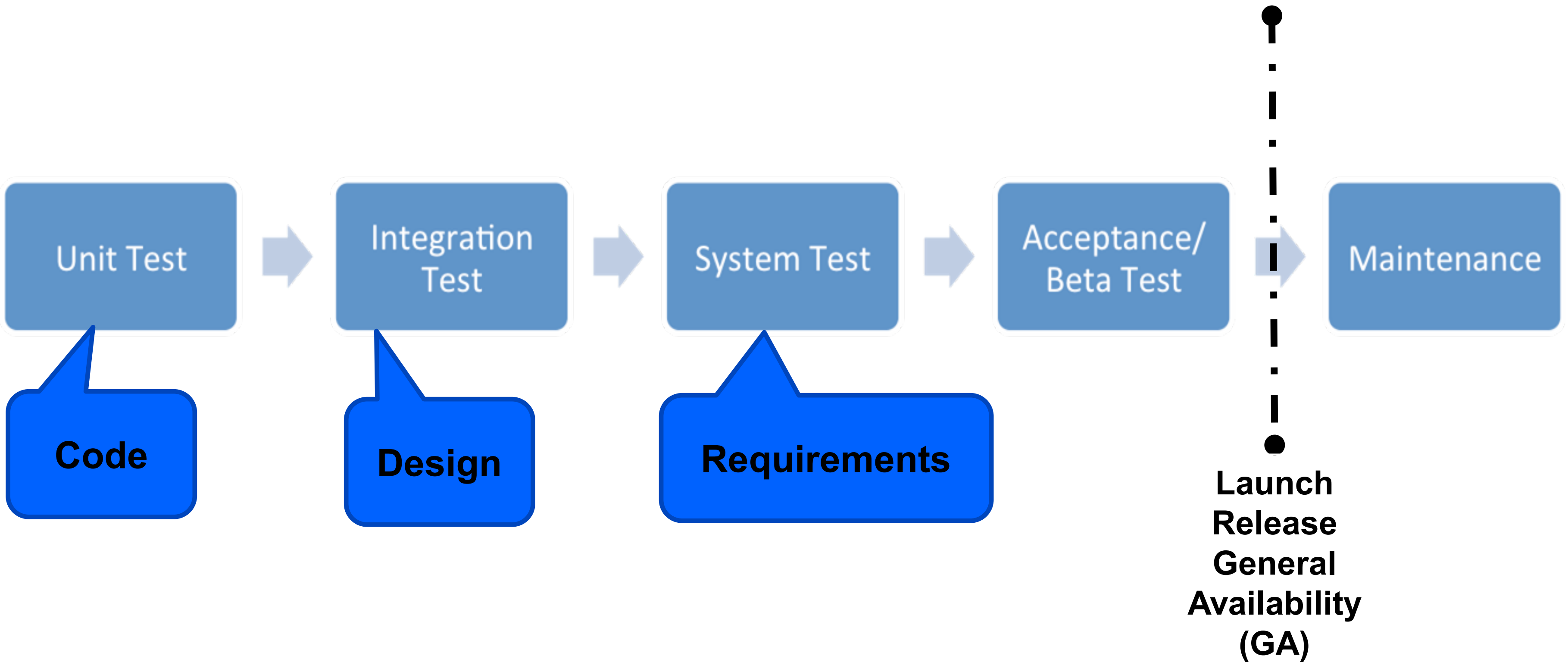
Test Phases



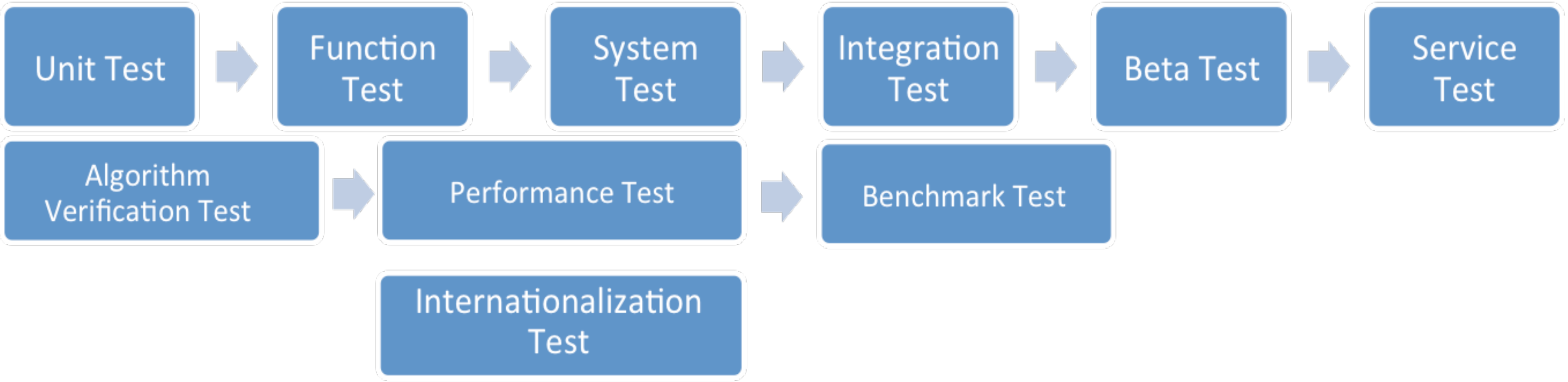
Non-Enterprise Test Pipeline Overview



Non-Enterprise Test Pipeline Overview



Enterprise Test Pipeline Overview



Unit Test

Function Test

System Test

Integration Test

Beta Test

Service Test

Algorithm Verification Test

Performance Test

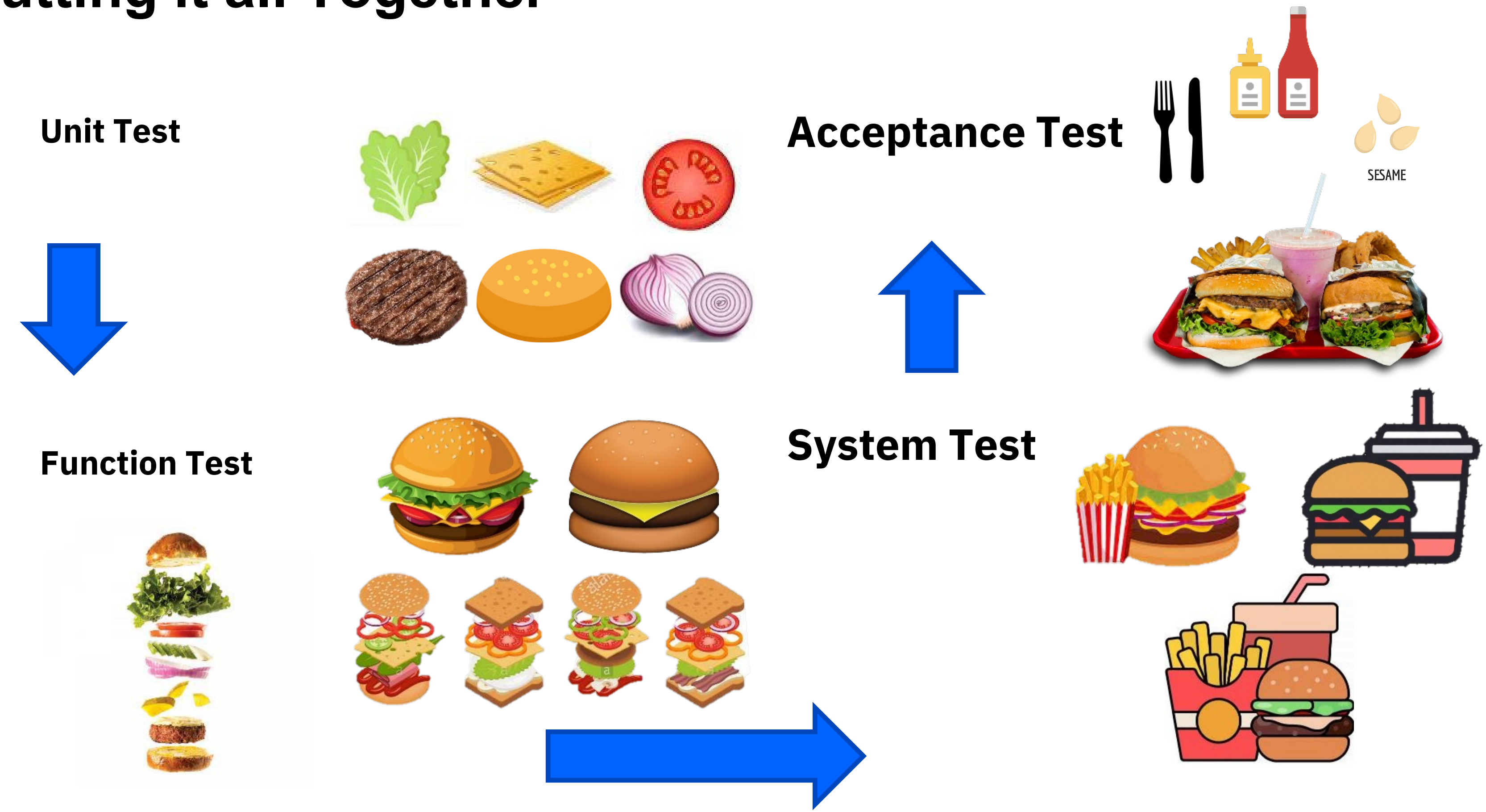
Benchmark Test

Internationalization Test

Common Activities Across Phases

- **Requirements Analysis**
  Understand customer requirements
- **Design Tests**
  Build and document your test strategy including automation
- **Test Plan Review**
  Inspect and approve test plans with subject matter experts and stakeholders
- **Execute Test**
  Perform tests and identify defects
- **Reflection**
  Analyze defects and test escapes

Burger Example
Putting it all Together



Unit Test

Function Test

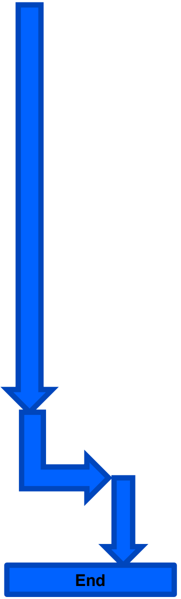
System Test

Acceptance Test

Unit Test – What is it?

- A unit typically refers to the smallest possible testable piece of the project (e.g. a line of code, a branch, function/method, etc.)
- In software, the goal is 100% line coverage
  - Taking all branches/paths in the code is acceptable
  - If 100% coverage isn’t possible, at least hit all new or changed lines
- Unit tests can be "less natural" with the developer or tester forcibly flipping bits or setting flags as needed to drive paths
  - Generally executed by the developer as a first pass at ensuring stability in their code, architecture, etc.

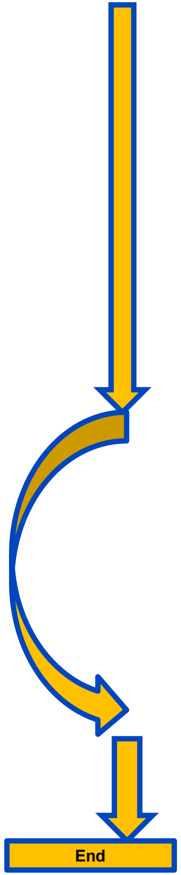
Unit Test – Coverage Example



```
def function1() {
  ! Define variables
  MyVar = 0 ! Variable to use in this program
  print("Starting variation")

  ! Call service
  MyVar = Proc1()

  ! Did the service work?
  If MyVar != 0 Then {
     ! Fail TC
     return("Something broke " + MyVar)
  }
  Else {
     return("Everything worked!")
  }
}
```



```
def function1() {
  ! Define variables
  MyVar = 0 ! Variable to use in this program
  print("Starting variation")

  ! Call service
  MyVar = Proc1()

  ! Did the service work?
  If MyVar != 0 Then {
     ! Fail TC
     return("Something broke " + MyVar)
  }
  Else {
     return("Everything worked!")
  }
}
```

Unit Test – Example

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_zero(self):
        Proc1() = {return 0;}    # stub
        self.assertEqual(function1(), 'Everything worked!')

    def test_nonzero(self):
        Proc1() = {return 1;}    # stub
        self.assertEqual(function1(), 'Something broke 0')
```

Function Test

- Functional and behavioral verification of interactions and integrations of units or components in a system

- Also known as integration test

Function Test

The goal is to validate all new features, behaviors and interactions as naturally as possible and to ensure no regressions in existing logic

- Mainline features and error paths
- Services
- Recovery logic
- Limits and boundaries are honored
- Serialization
- Counts
- Condition Codes

Function Test - Example

If we were to test a file system, what variations might we execute?

Test Description	Expected Results
Create a **new** file	File is created with a **size** of **0**
Create new file with **name conflict**	Appropriate return code
Modify a file that you **don't have authority** to change	File **contents** remain **unchanged**. Appropriate **abend** or **error code** is returned

…

Variations need to be as specific as possible on what function is being tested and what the expected outcome is

System Test

Focused on real world type usage such as how end users and customers would interact with the full system

- Ensures completeness of the system
- Verifies that no regressions occurred in the existing features
- Exposes problems in upgrade capabilities for customers looking to upgrade to a new version of the system
- Validates usability of features and components. If developers and testers have difficulty using the system, customers will too
- Uses customer-centric and native z/OS tools (e.g. IPCS, RMF, SDSF, etc.)

System Test - Example

- If we were to system test the same file system from earlier, which scenarios would we execute?

Test Description	Expected Results
Have a **thrasher** with **many users** accessing (reading, writing, modifying, etc.) the **file system**	**No** unexpected **abends or error codes**. File **contents** will be **consistent** and in a **healthy** state
Have users from **different OS** and **hardware** levels perform **file operations** against the **same file system**	**No** unexpected **abends or error codes**. File **contents** will be **consistent** and in a **healthy** state

- We want to focus on things like the interactions between different services, serialization and the number of users at different scales

Acceptance Test

Testing performed outside of the development team to verify customer requirements were implemented and behave accordingly

The goal is to verify that customer requirements were met…

- Using “customer-like” workloads
- Follows publications strictly
- Driving features like rolling IPLs
- Applying service
- Generally following “happy path” type tests, not injecting errors

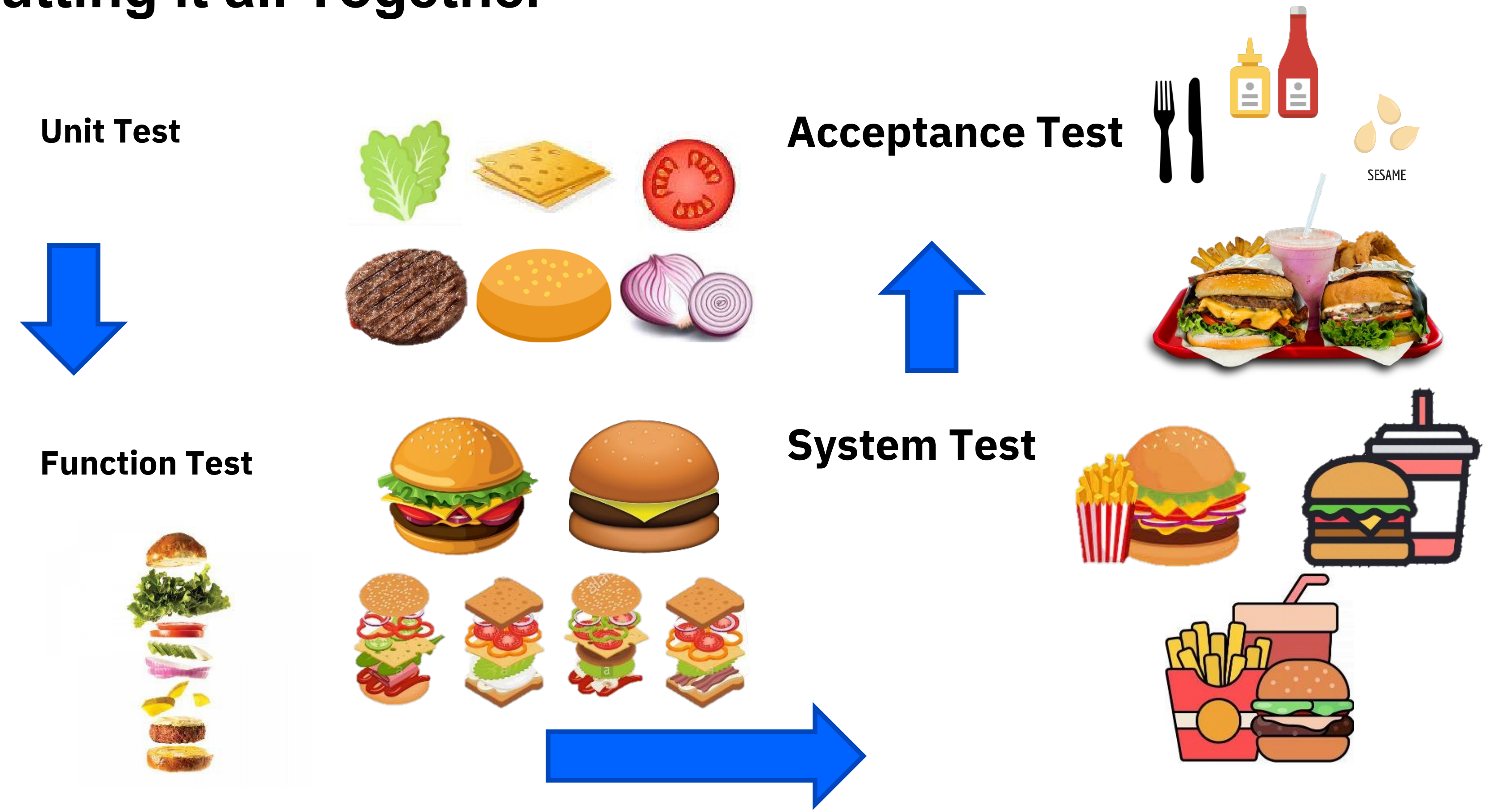
Acceptance Test – Example

- Returning to our file system example, how might we conduct an acceptance test from an end user's perspective?

Test Description	Expected Results
Have a **user** install a program which requires the creating and managing **files** using the **file system**	**No** unexpected **abends or error codes**. The program will install **successfully**
Have a user interface with a **text processing program** (e.g. Microsoft Word) to create a new document on the **file system** and print it out	**No** unexpected **abends or error codes**. File **contents** will be **consistent** and in a **healthy** state

- We want to focus on things like the end user experience and mainline functions performing as expected

Burger Example
Putting it all Together



Unit Test

Function Test

System Test

Acceptance Test

Questions?